SAVONIA

OPINNÄYTETYÖ - **AMMATTIKORKEAKOULUTUTKINTO**
TEKNIIKAN JA LIIKENTEEN ALA

SPV-SOVELLUS

Web-client

TEKIJÄ: Ari Liimatainen

SAVONIA-AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala	
Koulutusohjelma/Tutkinto-ohjelma Tietotekniikan koulutusohjelma	
Työn tekijä(t) Ari Liimatainen	
Työn nimi Spv-sovellus	
Päiväys 25.5.2020	Sivumäärä/Liitteet 22
Toimeksiantaja/Yhteistyökumppani(t) Wikli Group, Ari Liimatainen tmi	

Tiivistelmä

Opinnäytetyön tavoitteena oli kehittää Wikli Group Oy:lle sukupolvenvaihdossovellus. Spv-sovelluksen tarkoituksena oli korvata monimutkainen ja hankala käyttöinen Excel-taulukkokokonaisuus, jota käytettiin maatilan sukupolvenvaihdoksen suunniteluun ja veroseuraamuksien laskemiseen. Lisäksi sovelluksesta piti saada siirrettyä tietoja muihin yrityksen sovelluksiin ja toisin päin.

Projekti aloitettiin suunnittelemalla käyttötarkoitukseen sopiva sovellus yhteistyössä Wikli Group Oy:n työntekijöiden kanssa. Kun suunnitelmasta päästiin yhteisymmärrykseen, alettiin toteuttaa MVC-arkkitehtuurin mukaista web-client sovellusta, joka käytti SQL-tietokantaa tietojen tallennukseen.

Opinnäyteprojektissa valmistui sovellus, joka vastasi suunniteltua sovellusta. Sovelluksen jatkokehtys jatkuu edelleen ja se tullaan julkaisemaan.

Avainsanat
MVC, Web-client, SQL, Spv, Sovellus

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Information Technology			
Author(s) Ari Liimatainen			
Title of Thesis Spv Application			
Date	25 may 2020	Pages/Appendices	22
Client Organisation /Partners Wikli Group, Ari Liimatainen tmi			

Abstract

The aim of the thesis was to develop a generational change application for Wiklli Group Oy. The purpose of the Spv application was to replace the complex and cumbersome Excel spreadsheet entity which was used to plan a farm generation change and calculate tax concequences. In addition, data had to be transferred from the application to other applications in the company and vice versa.

The project was started by designing an application suitable for the intended use in cooperation with Wikli Group Oy's employees. When the plan was agreed upon, the implementation of the web-client application based on the MVC architecture, which used an SQL database to store data, was started.

An application that corresponded to the planned application was completed in the thesis project. The further development of the application will continue and it will be published in the future.

Keywords
Spv, Application, MVC, SQL, Web-client

SISÄLTÖ

1 JOHDANTO

Opinnäytetyön tavoitteena oli luoda MVC-arkkitehtuurin mukainen web-sovellus, joka korvaisi ja automatisoisi perinteisten Excel-talukoiden käytön.

Opinnäytetyössä toteutettiin Spv-sovellus, jossa käyttäjän tulisi helposti pystyä syöttämään maatilan tiedot ja sovellus laskisi mahdolliset veroseuraamukset, sekä miten sukupolvenvaidos kannattaisi tehdä.

Tilaajana tuotteelle oli alun perin suomalainen Wikli Group Oy, mutta suunnitelun jälkeen tilaajaksi vaihtui Ari Liimatainen tmi, joka on atk-palveluja tarjoava yritys.

Opinnäytetyö sisälsi sovelluksen suunnittelun, kehittämisen, testaamisen ja käyttöönoton. Suunnitteluvaiheessa pyrittiin saamaan selkeä kuva mitä haluttiin sovellukselta ja mihin kaikkeen sen täytyy pystyä, sekä yritettiin pitää sovellus mahdollisimman selkeänä ja helppokäyttöisenä. Kehitysvaiheessa luotiin sovellus ja kaikki tarvittavat osat sovelluksen toiminnan kannalta. Testaaminen tapahtui kehityksen mukana eli kaikki uudet toiminnot testattiin, sekä pidettiin sovellus toimivana koko kehityksen ajan. Käyttöönottoa ei oltu vielä opinnäytteen kirjoitus hetkellä tehty, vaan se on jätetty paremmin sopivaan ajankohtaan.

2 TERMIT JA LYHENTEET

Spv	sukupolvenvaihdos
MVC	Model-View-Controller. Käytetään usein web-sovelluksissa, joissa view on HTML sivu ja controller on koodi, joka kerää dataa ja luo sisällön HTML:lle. Model on itse sisältö, joka yleensä on tallennettu tietokantaan tai XML-tiedostoihin. (Microsoft)
.NET	.NET on kehitysalusta, johon on koottu työkaluja, ohjelmointikieliä ja kirjastoja monenlaisten ohjelmien työstämistä varten. Kun käytetään C#-, F#- tai Visual Basic-kieltä .NET tarjoaa loistavan alustan monenlaisille sovelluksille. .NET jakautuu kolmeen eri toteutukseen, jotka ovat .NET Core, .NET Framework ja Xamarin/Mono. Core on alustariippumaton toteutus, jolla voidaan tehdä esim. verkko-ohjelmia Windows:lle, Linux:lle ja macOS:lle. Framework tukee pelkästään Windows:ia ja Xamarin/Mono tukee suurimpia kannettavien laitteiden käyttöjärjestelmiä.
REST	Representational State Transfer, on HTTP-protokollaan perustuva arkkitehtuurimalli, jolla on mahdollista määritellä tiedonsiirto sovelluksesta toiseen.
EF	Entity Framework on avoimen lähdekoodin olio-relaatiomappaus, joka oli osa .NET Framework:ia versioon 6 asti.
code-first	Entity Framework esitteli code-first lähetymistavan versiossa 4.1, joka on pääasiallisesti hyödyllinen toimialue johtoisessa suunnittelussa. Lähestymistavassa luodaan luokat, joista EF muodostaa tietokannan.
CRUD	Tulee sanoista create, read, update ja delete, jotka ovat neljä perustoimintoa tietojenhallinnassa.

3 KÄYTETYT TEKNIIKAT

Tässä käydään läpi opinnäytetyössä tuotetun Spv-sovelluksen kehittämiseen tarvittavat tekniikat ja ohjelmistot.

3.1 Visual Studio 2017

Visual Studio on Microsoftin luoma ohjelmankehitysympäristö, jossa voi käyttää useita ohjelmointikieliä, kuten esimerkiksi C#. Ohjelmistolla voidaan luoda esimerkiksi Windows-, web, ja mobiilisovelluksia ja siihen voidaan integroida monien eri valmistajien täydennyksiä.

3.2 C#

Englannin kielen lausunta C sharp. C# on yksinkertainen ja olio-orientoitunut korkeamman tason ohjelmointikieli, joka on julkaistu kesäkuussa 2000. C# kehitettiin, koska haluttiin ohjelmointikieli, joka yhdistäisi MS Visual Basicin tuottavuuden ja C++:n tehokkuuden. (Sivonen, 2004)

3.3 HTML

Lyhenne sanoista HyperText Markup Language. HTML on avoimesti standardoitu kuvauskieli, jolla kuvataan hyperlinkkejä sisältävää tekstiä. HTML erityisesti tunnetaan kielenä, jolla internetsivut on tehty. (W3C)

3.4 SQL

Lyhenne sanoista Structured Query Language. SQL on IBM:n kehittämä standardoitu kyselykieli, jolla voidaan tehdä erilaisia hakuja, muutoksia ja lisäyksiä relaatiotietokantaan. Kieli kehitettiin 1970-luvulla alun perin System R -tietokantajärjestelmää varten. (Helsingin yliopisto)

3.5 Microsoft Azure

Microsoft Azure on Microsoftin kehittämä pilvipohjainen palvelu, joka on tehty sovellusten ja palveluiden rakentamista, testausta, julkaisemista ja hallintaa varten. Se tukee monia erilaisia ohjelmointi kieliä, työkaluja ja viitekehyksiä, sisältäen Microsoft pohjaisia ja kolmannen osapuolen sovelluksia ja palveluita. Azure esiteltiin ensimmäisen kerran 2008 nimellä ”Project Red Dog” ja julkaistiin 1. helmikuuta 2010 nimellä ”Windows Azure”, nimeä muutettiin edelleen 2014 ”Microsoft Azure”:ksi. (Microsoft)

4 SUKUPOLVENVAIHDOKSEN LASKEMINEN

Yrityksessä käytettiin sukupolvenvaihdoksen laskemiseen Excel-taulukoista koostunutta kokonaisuutta, jossa kaikki laskentaan vaadittavat kaavat oli sijoiteltu eri välilehdille. Kaavoihin piti syöttää tiedot manuaalisesti ja tarvittavia tietoja piti etsiä yrityksen muista sovelluksista, sekä verohallinnon verotusohjeista. Kokonaisuus oli epälooginen ja vaati paljon opettelua, jotta käyttö olisi sujuvaa.

Maatilan sukupolvenvaihdoksen laskemisessa huomioon otettavia asioita olivat mm. maatilan omistajien ikä vaihdoshetkellä, maatilan omaisuus, sekä verottajan määrittämät verotus arvot maatilan omaisuudelle. Maatilan omaisuudeksi laskettiin kaikki asuinrakennuksesta tuotanto eläimiin ja kaikille piti manuaalisesti etsiä verottajan valmiiksi määrittämät arvot, näihin arvoihin vaikutti myös tilan sijainti kunta.

Suurimpia ongelmakohtia olivat tietojen syöttämisen monimutkaisuus, tietojen jakaminen ja taulukoiden muokattavuus. Samojen tietojen manuaalinen syöttäminen monelle eri välilehdelle oli työlästä ja mahdollisti näppäily virheet. Tietojen jakamaninen työryhmän kesken oli haasteellista, koska koko kokonaisuus piti lähettää sitä tarvitseville esim. sähköpostin liite -tiedostoina. Excel-taulukot olivat melko haasteellinen käyttöympäristo senkin suhteen, että kaikki taulukoita käyttävät käyttäjät muokkasivat niitä parhaaksinäkemällään tavalla, joka aiheutti erilaisten versioden leviämisen ja aiheutti tiedonsiirto ristiriitoja ja monimutkaisti lisää jo haasteelllista kokonaisuutta.

5 OHJELMISTON SUUNNITTELU

Projektin alkuvaiheessa määriteltiin lähtökohdat sovellukselle, jotta saataisiin mahdollisimman useat aiemman järjestelmän ongelmat poistetuksi tai helpotettua työskentelyä. Tähän vaiheeseen kului huomattavasti enemmän aikaa kuin olisi pitänyt, tästä johtui, että suunniteltu aikataulu jäi haaveeksi.

5.1 Vastaavat järjestelmät

Projekitin alussa otettiin selvää, miten yritykset hoitivat suomessa sukupolvenvaihdoksen laskennan. Wikli Group teki paljon yhteistyötä muiden asiantuntijayritysten kanssa ja saatiin tietää, että kaikki sukupolvenvaihdoksien veroseuraamuksia laskeneet yritykset pyörittelivät samankaltaisia Excel-taulukoita. Ilmeisesti mitään vastaavaa sovellusta ei ollut tehty tai se oli niin hyvin piilotettu, että siitä ei kerrottu ulkopuolisille. Päädyttiin tulokseen, ettei vastaavaa järjestelmää ollut saatavilla projektin aloitushetkellä.

5.2 Käyttötarkoitus

Toimeksiannon tarkoituksena oli kehittää sovellus, joka yksinkertaistaisi ja helpottaisi maatilan sukupolvenvahdoksen veroseuraamusten laskemista. Sovelluksen pitäisi korvata usean Excel-taulukon aihettama sekava laskentakaava, sekä uusien tietojen syöttäminen, muuttaminen ja poistaminen tapahtuisi helpommin. Sovelluksen pitäisi olla kaikkien yrityksen työntekijöiden käytettävissä ja useat käyttäjät voisivat käyttää tarvittaessa samoja tietoja yhtäaikaisesti, sekä tietoja pitäisi helposti saada tuotua sovellukseen ja vietyä muihin sovelluksiin.

Alustavana suunnitelmana oli tehdä web-client sovellus, jolle tehtäisiin RESTful-rajapinta tietojen siirtämiseen ja tiedot tallennettaisiin tietokantaan. Näin sovellus olisi kaikkien yrityksen työntekijöiden käytettävissä, tiedonsiirrot onnistuisivat rajapinnan kautta sovelluksien välillä ja tiedot tallentuisivat yhteiseen tietokantaan, jossa ne olisivat kaikkien käytettävissä.

5.3 Tekniikoiden valinta

Valittaessa tekniikoita pyrittiin pitämään sovellus helposti työstettävänä ja mahdollisimman yksinkertaisena, koska sovellus tulisi käyttöön yritykseen, jossa oli vain vähäistä osaamista C# kieleen. Projektin alussa annettiin ns. vapaat kädet tekniikoiden valintaan, mutta suunnittelun edetessä huomattiin tilaajan vaativan melko rautaisella kädellä tiettyjen tekniikoiden käyttöä.

5.3.1 Kehitysympäristö

Lähtökohtaisesti sovellusta lähdettiin kehittämään Visual Studio-ympäristöön, koska se tarjosi hyvät mahdollisuudet monenlaisien sovelluksien kehitykselle. Visual Studio tarjosi hyviä työkaluja ohjelmoinnin helpottamiseksi esim. ”Intellisense”-toiminnon eli sisäänrakennetut vihjeet ja kuvaukset, sekä automaattinen rivien täyttö nopeamman ohjelmoinnin saavuttamiseksi.

5.3.2 Ohjelmointikieli

Ohjelmointikieltä valittaessa haluttiin pitää koodin kirjoitusasu mahdollisimman yksinkertaisena, jotta ohjelman jatkokehitys olisi helpompaa. Tilaajan microtukihenkilö ei osannut mitään ohjelmointikieltä, mutta vannoi opettelevansa ainakin C#-kielen tarvittaessa. Vaatimuksiin vedoten parhaimmaksi kieleksi päädyttiin C#-kieleen. Vaikka C# olisikin korkeamman tason ohjelmointikieli sitä olisi silti suhteellisen helppoa lukea ja suurinosa monimutkaisista tehtävistä, kuten muistin hallinta olisi jätetty .NET:n hoidettavaksi. Se olisi myös staattisesti kirjoitettu kieli eli koodi tarkistettaisiin ennen kuin se muutetaan ohjelmaksi, tämä mahdollistaisi helpomman virheiden etsinnän.

5.3.3 Tietokanta

Tietokanta päätettiin toteuttaa SQL:llä, koska toimeksiantajalla oli jo toiminnassa oleva SQL-palvelin. Näin sovelluksen käyttöönotto ei vaatisi suuria muutoksia yrityksen järjestelmiin. Tietokanta yritettiin pitää mahdollisimman pienenä ja yksinkertaisena, jossa tosin onnistumatta. Tietokannan suunnitteluun meni paljon aikaa, mutta lopulta saatiin suhteellisen yksinkertainen tietokanta.

5.4 Käyttöliittymä

Tarkoituksena oli luoda selkeä ja yksinkertainen käyttöliittymä, jossa olisi selkeät paikat kaikille tiedoille. Käyttöliittymä mahdollistaisi tietojen helpon syöttämisen, selaamisen, muokkaamisen ja poistamisen. Käyttöliittymässä pitäisi olla näkymät jokaiselle tietokannan taululle ja yhteenveto näkymä, jossa näkyisi valmiiksi lasketut veroseuraamukset, sekä ne voisi tulostaa tai tallentaa tietokoneelle. Näkymiin tulisi myös tuo ja vie toiminnot tietojen siirtoa varten.

5.5 Rajapinta

Toimeksiantajan toiveesta päädyttiin RESTful-rajapinnan käyttöön, joka käyttäisi yksinkertaisia HTTP kyselyjä kommunikointiin. Käytettäessä HTTP protokollaa REST olisi vain yksinkertainen pyydä/vastaa mekanismi sovellusten välillä. REST oli hyvä myös siinä mielessä, ettei se rajoittanut päätelaitteiden käyttöä vaan se toimii niin iOS:lla kuin Windows:lla, eli vaikka sovellus tulisikin windows:lle voisi iOS:lla hakea sovelluksesta tietoa rajapinnan kautta.

6 OHJELMISTON TOTEUTUS

Ohjelmiston toteutuksessa kerrotaan tarkemmin suunnitteluvaiheessa päätettyihin ratkaisuihin ja miten ratkaisut toimivat spv-sovelluksessa.

6.1 Ohjelmoinnin aloitus

Pitkään kestäneen suunnittelun jälkeen päästiin aloittamaan itse ohjelmiston toteuttamista, hyvin suunniteltu on puoliksi tehty, ei ollut. Alkuvaiheessa suunniteltu tietokanta osoittautui puutteelliseksi ja vaati lisää hiomista, joka aiheutti ongelmia aikataulun suhteen. Kun tietokannan suunnitelma oli saatu tarpeeksi laajaksi, aloitettiin ohjelman tekeminen. Visual Studio antoi mahdollisuuden aloittaa projektin niin, että se sisälsi jo valmiiksi MVC-mallin.

Kun projekti oli virallisesti saatu aloitetuksi luotiin tietomalli, sekä tietokannan kontekstiluokka. Tietokanta kontekstissa määriteltiin mitkä luokat kuuluivat tietomalliin, toisin sanoen mitkä luokat muodostaisivat tietokannan. Tässä vaiheessa todettiin, ettei koko tietomallia voida luoda kerralla, koska EF ei osannut luoda tietokantaa tietomallin pohjalta, jossa oli ”liikaa” luokkia ja viittauksia niiden välillä. EF olisi osannut luoda tietokannan, jos tietomallissa olisi ollut vain yksi viittaus yhteen kenttään, mutta kokonaisessa tietomallissa viitattiin useasta luokasta yhden luokan id kenttään. Tästä viisastuneena luotiin aluksin vain osa tietomallista eli kolme luokkaa, joista yksi oli ”Farmstead”-luokka. Edellä mainittu ongelma olisi saatu korjattua ottamalla ”System.ComponentModel.DataAnnotations”-nimiavaruus käyttöön ja määrittämällä viittaukset manuaalisesti, mutta käytettiin oikotietä ja nimiavaruus otettiin käyttöön vasta paljon myöhemmin.

Ohjelman testaamiseksi tarvittiin testitiedot, jotka syötettiin tietokantaan. Luotiin alustusluokka, joka poistaisi ja loisi uuden tietokannan aina, jos tietokannan konteksti muuttuisi, sekä syöttäisi testitiedot tietokantaan. Ohjelman ollessa vielä niin alussa helpointa oli testata tietokannan toimivuus paikallisella tietokannalla.

Kun tietokanta todettiin toimivaksi, tehtiin ensimmäinen ohjain ja näkymä. Tähänkin Visual Studio tarjosi vaihtoehdon, jolla pystyi luomaan luokan ohjaimen ja näkymän käyttäen EF:ia. Kun saatiin tarkistettua ohjaimen koodi, testattiin lähtisikö sovellus edes käyntiin. Käynnistyttyään testattiin, että näkymä näytti tietokannassa olleet testitiedot, hyvin toimi. Samalla todettiin, että ”Read”-metodi toimi, kun tietokannasta saatiin luettua tiedot.

6.2 CRUD-toiminnot

Sitten alettiin muokkaamaan aiemmin luotua ”Farmstead”-ohjainta, ”Create”-metodiin lisättiin ”try-catch”-osio, joka yrittäisi luoda uuden mallin ja jos se ei siinä onnistuisi pyydystäisi se virheilmoituksen. Näin saatiin tehtyä yksinkertaisen virhelokin pohja, josta lisää myöhemmin. Testattiin ”Create”-metodia syöttämällä virheelliset tiedot ”lisää uusi”-näkymään ja sitten oikeat tiedot, todettiin toimivaksi.

”Update”-metodia muutettiin niin, että se vastaanottaisi id:n ja tarkistaisi onko id ”true”, jos ei olut näytettiin virheilmoitus, jos id oli ”true” muokattiin id:n osoittamaa riviä ja ”try-catch”-osiolla tarkistettiin, onnistuiko muutoksien tallennus tietokantaan. Testattiin syöttämällä väärä id ja saatiin virheilmoitus ”bad request”, sitten syötettiin oikea id ja saatiin muokattua kyseistä riviä.

”Delete”-metodeja lisättiin kaksi, toinen hoiti virheentarkistuksen ja virheilmoituksen, joka tarkisti poistettavan tiedon olevan olemassa, sekä tiedon poisto tallentuisi tietokantaan. Toinen taas toteutti varsinaisen tietojen poiston tietokannasta. ”Poista”-näkymään tehtiin pieni varoitusteksti, joka pyysi vahvistamaan poiston. Testattiin tämäkin syöttämällä väärä id, jolla saatiin ”bad request” virheilmoitus, ”savechanges” virheilmoitus saatiin poistamalla tietokantayhteys poiston aikana. Rivin poistokin onnistui hyvin.

Tietokantayhteyden sulkeminen olisi järkevintä, kun se tehtäisiin heti kun tietokantaa ei enää tarvittaisi, tämä vapauttaisi resurssit sovelluksen muihin käyttötarkoituksiin. Tietokanta yhteyden sulkeminen toteutettiin muokkaamalla ohjaimen IDisposable rajapintaa koodilla, joka korvasi Dispose metodin ja poisti instanssin.

6.3 Lajittelu- ja hakutoiminnot, sekä sivutus

Sovelluksen laajentuessa olisi hyvä, kun tietoja voitaisiin lajitella ja hakea jotenkin. Siispä alettiin tekemään lajittelutoimintoa. Aiemmin muokattuun ohjaimeen lisättiin uusi viewresult nimeltään ”index”, joka ottaisi vastaan sortOrder muuttujan, jota käytettäisiin lajittelun toteutuksessa.

```
switch (sortOrder)
{
    case "id_desc":
        farmsteads = farmsteads.OrderByDescending(f => f.Id);
        break;
    case "name":
        farmsteads = farmsteads.OrderBy(f => f.Name);
        break;
    case "name_desc":
        farmsteads = farmsteads.OrderByDescending(f => f.Name);
        break;
    default:
        farmsteads = farmsteads.OrderBy(f => f.Id);
        break;
}
```

Kuva 1 Lajittelutoiminnon määrittäminen

Kuva 1 esittää, miten ”switch-case”-rakenteella saatiin vaihdettua lajittelujärjestys nimen tai id:n mukaan laskevaksi tai nousevaksi, alkuperäinen järjestys olisi siis id:n mukaan nouseva. ”SortOrder”-muuttuja saisi arvon sen mukaan, mitä sarakkeen nimeä käyttäjä klikkaisi.

```
<tr>
    <th>
        @Html.ActionLink("Id", "Index", new { sortOrder = ViewBag.IdSortParm, currentFilter = ViewBag.CurrentFilter })
    </th>
    <th>
        @Html.ActionLink("Nimi", "Index", new { sortOrder = ViewBag.NameSortParm, currentFilter = ViewBag.CurrentFilter })
    </th>
    <th>
        Muokattu
    </th>
    <th>
        Luotu
    </th>
    <th></th>
</tr>
```

Kuva 2 Lajittelutoiminnon määrittäminen näkymään

Sarakkeiden nimet muutettiin linkeiksi (Kuva 2), joita klikkaamaalla sortOrder-muuttuja saisi uuden tiedon järjestyksestä ja ohjaimen ”switch-case”-rakenne vaihtaisi lajittelujärjestystä. Samoilla menetelmillä saataisiin lajittelutoiminto niin moneen sarakkeeseen, kuin olisi tarvinnut. Sitten testattiin lajittelutoiminto ja todettiin se toimivaksi.

Seuraavaksi aloitettiin hakutoiminnon toteutus, jonka tarkoitus oli olla hakukenttä, johon syötettiin maatilan nimi tai nimen osa. Hakutoiminnon toteuttamiksi jouduttiin lisäämään ”index” viewresult:iin uusi vastaanotettava muuttuja ”searchString”, joka kertoisi ohjaimelle olisiko haku jo tehty vai edettäisiinkö vakioarvolla.

```
if (!string.IsNullOrEmpty(searchString))
{
    farmsteads = farmsteads.Where(f => f.Name.Contains(searchString));
}
```

Kuva 3 Yksinkertainen hakutoiminto

Kuva 3 esittää yksinkertaisen hakutoiminnon ja virheen korjauksen, jossa if-lause tarkistaisi olisiko ”searchString”-muutuja hyväksyttävä hakuehdoksi, jos olisi verrattaisiin muuttujaa tietokannassa oleviin maatilojen nimiin ja näytettäisiin ne näkymässä. Hakukentän sai näkyviin näkymään lisäämällä ”cshtml”-tiedostoon html beginform:in, joka loi valmiin kentän, jolle määriteltiin, että se lähettäisi ”searchString”-muuttujan siitä mitä kenttään kirjoitettaisiin, kun haku nappia painettaisiin. Todettiin testaamalla hakutoiminnon toimivuus, niin koko nimillä kuin nimenosillakin.

Sivujen tekoon vaadittiin uusi NuGet paketti nimeltään ”PagedList.MVC”, joka saatiin ladattua helposti NuGet Package Manager Consolesta. Kirjoitettiin vain consoleen ”Install-Package PagedList.MVC” ja Visual Studio hoiti loput, ainoastaan piti rakentaa projekti uudelleen. Sitten päästiin taas

ohjaimen ”index”-osion kimppuun, osioon piti lisätä muuttujia, joihin tiedot tuodaan näkymästä. Ohjaimeen tehtiin tarkistus, jossa tutkittasiin oltaisiko sivustolta haettu jo jotain, jos olisi näytettäisiin haetut tiedot, muutoin näytettäisiin ensimmäinen sivu. Ohjaimesta saatiin valittua, montako riviä tietoa kullakin sivulla näytettiin, koska haluttiin testata niidenkin toimivuus, määritettiin kolme riviä per sivu.

Näkymälle piti kertoa, että se käyttäisi ”PagedList”-pakettia ja määritellä hakujen, sekä tietojen järjestyksen tallentuminen muuttujiin, joita ohjain käyttäisi. Näkymään piti myös tehdä paikka sivunumeroille ja napit sivuilla navigointia varten, kun ne saatiin tehtyä, päästiin testaamaan. Nyt testattiin niin sivujen, lajittelun ja hakujenkin toiminta, koska tietojen piti liikkua näkymän ja ohjaimen välillä. Saatiin todettua, että kaikki toimi halutulla tavalla.

6.4 Sovelluksen julkaiseminen valmistelu ja virhelokin käyttöönotto

Kun sovellus julkaistaisiin Azureen, paikallinen tietokantakin julkaistaisiin Azure SQL Database:n, joka oli pilvitietokantapalvelu. Julkaisu pilveen aiheuttaisi useammin väliaikaisia yhteysongelmia kuin käyttäessä paikallista web-palvelinta ja tietokantaa. Useimmat yhteysongelmat kuitenkin olisivat vain väliaikaisia ja korjautuisivat itsestään, kun kysely lähetettäisiin uudestaan. Käyttömukavuuden kannalta piti keksiä, miten kyselyn uudelleenlähetys tehtäisiin niin ettei käyttäjälle tulisi siitä turhaa päänvaivaa. Ongelmaan tarjosi vastauksen EF versio 6:sta löytynyt ”persistent connection”-toiminto, joka automatisoi kyselyn uudelleenlähetyksen yhteysvirheen tapahtuessa. Toiminnon käyttöönotto tapahtui luomalla uusi ”FarmerConfiguration”-luokka, jolle perittiin ominaisuudet ”DbConfiguration” aliluokalta ja lisäämällä ohjaimeen ”System.Data.Entity.Infrastructure;”. Luotu luokka määritti uuden SQL strategian, jolla lähetettiin kysely useita kertoja, jolloin vasta usean epäonnistuneen yrityksen jälkeen annettiin virheilmoitus.

Kun uudelleenlähetyskäytäntö oli otettu käyttöön, todettiin käytännön testaus haastelliseksi, koska yhteysvirheitä ei kovin usein tapahtunut varsinkaan, kun ajettiin ohjelmaa paikallisesti. Toiminnon testaamiseksi tarvittiin tapa keskeyttää kyselyt, jotka EF lähetti SQL palvelimelle ja korvata SQL palvelimen normaalivastaus poikkeus tyypillä.

Parempi tapa virhelokin tekemiseksi oli käyttää mieluummin käyttäjäliittymää, kuin kovakoodata kyselyt ”loki”-luokkaan. Tämä mahdollisti helpomman lokin mekanismien muuttamisen myöhemmin. Niimpä luotiin ”ILogger”-käyttäjäliittymäluokka, johon määritettiin metodit informaatiolle, varoitukselle ja virheelle, sekä ”TraceApi”-metodi, joka mahdollisti kyselyjen viiveen seuraamisen ulkoisiin palveluihin kuten SQL tietokantaan. Seuraavaksi luotiin ”Logger”-luokka, joka peri ominaisuudet ”ILogger”-käyttäjäliittymäluokalta ja käytti ”System.Diagnostics” jälityksessä, joka on .NET:in sisäänrakennuttu toiminto, jolla pystyttiin helposti generoimaan ja käyttämään jälitystietoja.

Seuraavaksi luotiin pysäytys luokat, jotka EF kutsui aina kun se lähetti kyselyn tietokannalle, yksi simuloi yhteysongelmia ja toinen kirjasi lokia. Ensin luotiin ”FarmerInterceptorLogging”-luokka, joka peri ominaisuudet ”DbCommandIntercerptor”-luokalta ja syrjäytti sen, kun kyselyt ja komennot on-

nistuivat, koodi kirjoitti tietolokia ja poikkeuksissa se loi virhelokin. Sitten luotiin ”FarmerInterceptorTrancientErrors”-luokka, jonka tarkoituksena oli tehdä ”tyhmyrivirhe” ja palauttaa verkkovirhettä vastannut virhekoodi. Jotta luokat saatiin toimimaan, piti ne lisätä ”Application_Start ()”-metodiin, vaihtoehtoisesti ne voisi lisätä mihin vain koodiin, mutta selkeyden vuoksi ne sijoitettiin em. metodiin.

Sitten päästiin testaamaan lokia ja uudelleen lähetystä, kun sovellus käynnistettiin, lokiin alkoi virrata SQL kyselyjä. Tämä johtui siitä, että EF käynnistyessään tarkisti tietokanta version ja migraatio historian, migraatioita ei ohjelmistossa vielä ollut. Kun sovellus käynnistyi ja hakukenttään syötettiin tyhmyri, huomattiin selaimen odottavan muutaman sekunnin ajan, jonka aikana EF yritti lähettää kyselyn useaan kertaan. Loppuksi virhelokiin ilmestyi virheilmoitus ”tyhmyri” useaan kertaan, näin voitiin todeta kaiken toimivan niin kuin haluttiin.

6.5 Ensimmäinen Entity Framework -migraatio

Koska kehitettiin uutta sovellusta, tietomalli muuttui toistuvasti ja aina kun malli muuttui, se ei ollut enää yhtenevä tietokannan kanssa. EF oli konfiguroitu poistamaan ja luomaan uusi tietokanta aina, kun tietomalliin tuli muutoksia. Kun lisättiin, poistettiin tai muokattiin tietomallin luokkia, seuraavalla käynnistyksellä sovellus automaattisesti poisti olemassa olleen tietokannan ja loi uuden vastaamaan tietomallia, sekä syötti testitiedot tietokantaan. Tämä metodi piti tietokannan yhtenevänä tietomalliin ja se toimi hyvin kehityksessä siihen asti, että sovellus julkaistaisiin tuotantoon. Kun sovellus olisi käytössä tuotannossa, se yleensä tallentaisi tietoa, joka haluttaisiin pitää. Tietoja ei haluttaisi menettää, joka kerta kun tietokantaan lisättäisiin esimerkiksi sarake. ”Code first migration” ominaisuus ratkaisi edellä mainitun ongelman, sallimalla code-first:in päivittää tietokannan mallia, poistamatta ja luomatta tietokantaa aina uudelleen.

Aluksi poistettiin ”Web.config”-tiedostosta ”context”-elementti, joka aiemmin hoiti tietokanna luomisen, poistamisen ja tietojen syöttämisen. Samaiseen tiedostoon asetettiin uusi ”connectionString”-elementti, jotta ensimmäinen migraatio loisi uuden tietokannan ja voitaisiin todeta migraation onnistuneen. Sitten syötettiin Package Manager Console:en komento ”enable-migrations”, joka loi projektiin kansion ”Migrations”, ja sijoitti sinne ”Configuration”-tiedoston, mitä muokkaamalla pystyttiin konfiguroimaan migraatioita. Seuraavaksi ajettiin komento ”Add-Migration InitialCreate”, joka loi tyhjän migraation, jossa kuitenkin oli kuva nykyisestä tietomallista. Kuva sisälsi ”Up”-metodin, joka loisi tietokannan taulut data mallin mukaan, sekä ”Down”-metodi, jolla taulut poistettiin.

Testitietojen syöttäminen ei enää onnistunut aiempaa metodia käyttämällä, koska ”context”-elementti oli poistettu. Siirryttiin käyttämään ”Seed”-metodia, joka syöttäisi testi tiedot aina kun tietokanta luotaisiin tai sitä päivitettäisiin. Metodin käyttö ei ollut vaadittavaa, koska migraatiot päivittävät tietokannan hävittämättä tietoja. ”Seed”-metodilla saatiin helposti syötettyä kaikki testitiedot uuteen tietokantaan, metodi sijoitettiin ”Configuration”-tiedostoon. Kokemuksesta viisaantuneena ristiriitojen välttämiseksi ”Seed”-metodin täytyi syöttää tiedot tietokantaan ”AddOrUpdate”-metodilla,

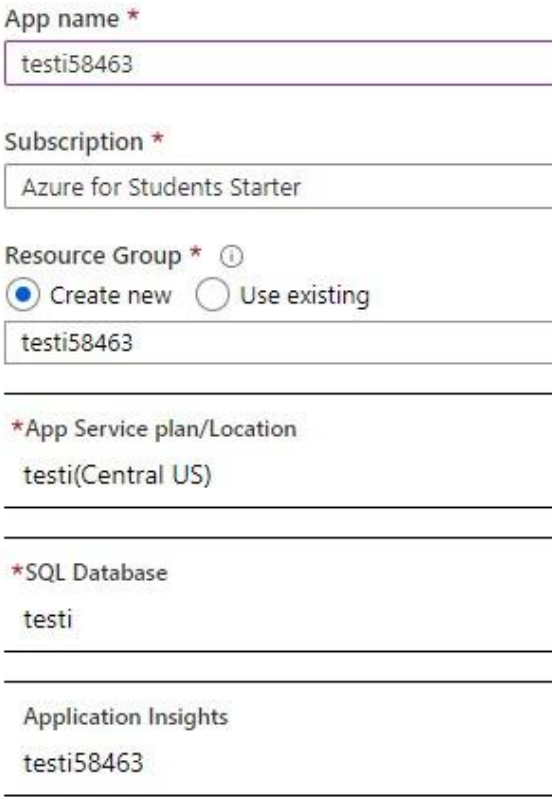
koska tietokannan päivityttyä yritti ”Seed”-metodi syöttää tiedot tietokantaan, jossa ne jo olivat. Näin vältyttiin virheilmoitukselta.

Testattiin olisiko migraatio onnistunut halutulla tavalla, tämän toteamiseksi Package Manager Console:en ajettiin komento ”update-database”. Update-database komento ajoi ”Up”-metodin ja loi tietokannan, jonka jälkeen se ajoi ”Seed”-metodin. Sama prosessi toimii automaattisesti tuotannossa, kun sovellus julkaistaan. Varmistettiin vielä ”Server Explorer”:lla, että tietokantayhteys toimi ja tietokannassa olisi testitiedot, todettiin toiminnot onnistuneiksi.

6.6 Julkaisu Microsoft Azureen

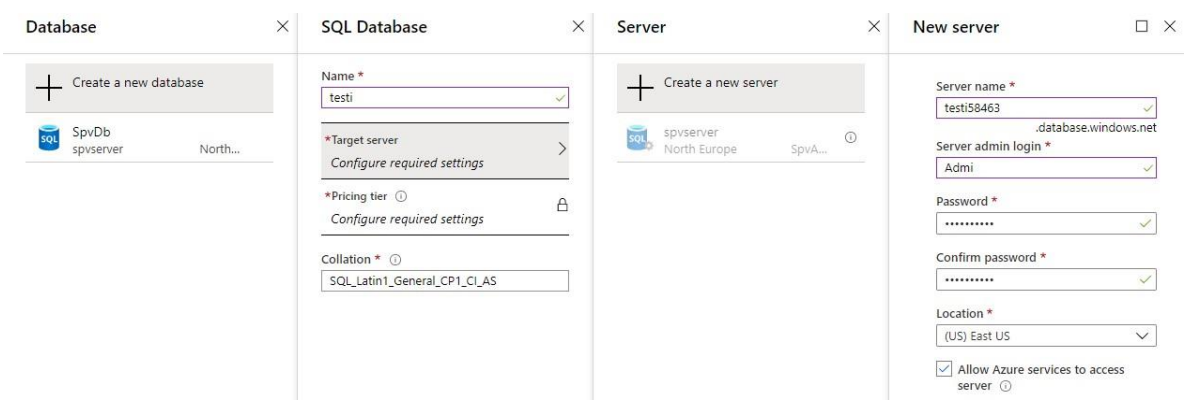
Tähän asti sovellus oli toiminut paikallisesti kehitys tietokoneella, jotta muutkin ihmiset voisivat käyttää sovellusta internetin välityksellä täytyisi sovellus julkaista isäntäpalvelimelle. Tässä projektissa käytettiin Azurea, joka vaati käyttäjätunnuksien luonnin, joiden tekeminen on täysin ilmaista. Savonian käyttäjätunnuksiin oli jo liitetty Azuren käyttöoikeudet siispä käytettiin koulun tunnuksia sovelluksen julkaisemiseen Azureen. Kun sovellus julkaistaisiin Azureen sitä ajettaisiin jaetulla isäntäpalvelimella, joka tarkoittaa virtuaalipalvelinta, mikä on jaettu muiden Azure käyttäjien kesken. Jaettu isäntäpalvelin on kustannustehokas tapa tutustua pilviympäristöön. Jos tulevaisuudessa sovelluksen verkkoliikenne kasvaisi olisi helppoa siirtää sovellus omalle virtuaalipalvelimelle, joka vastaisi paremmin sovelluksen tarpeita.

Julkaisua varten valmisteltiin Azuren pilviympäristö toimimaan sovelluksen uutena julkaisuympäristönä. Ensimmäiseksi luotiin ”Azure Management Portal”:ssa uusi resurssi ja valittiin ”Web App + SQL”, sekä ”Everything”, viimeisenä valittiin ”Create”, jolloin avautui ikkuna uuden resurssin luomiseksi.



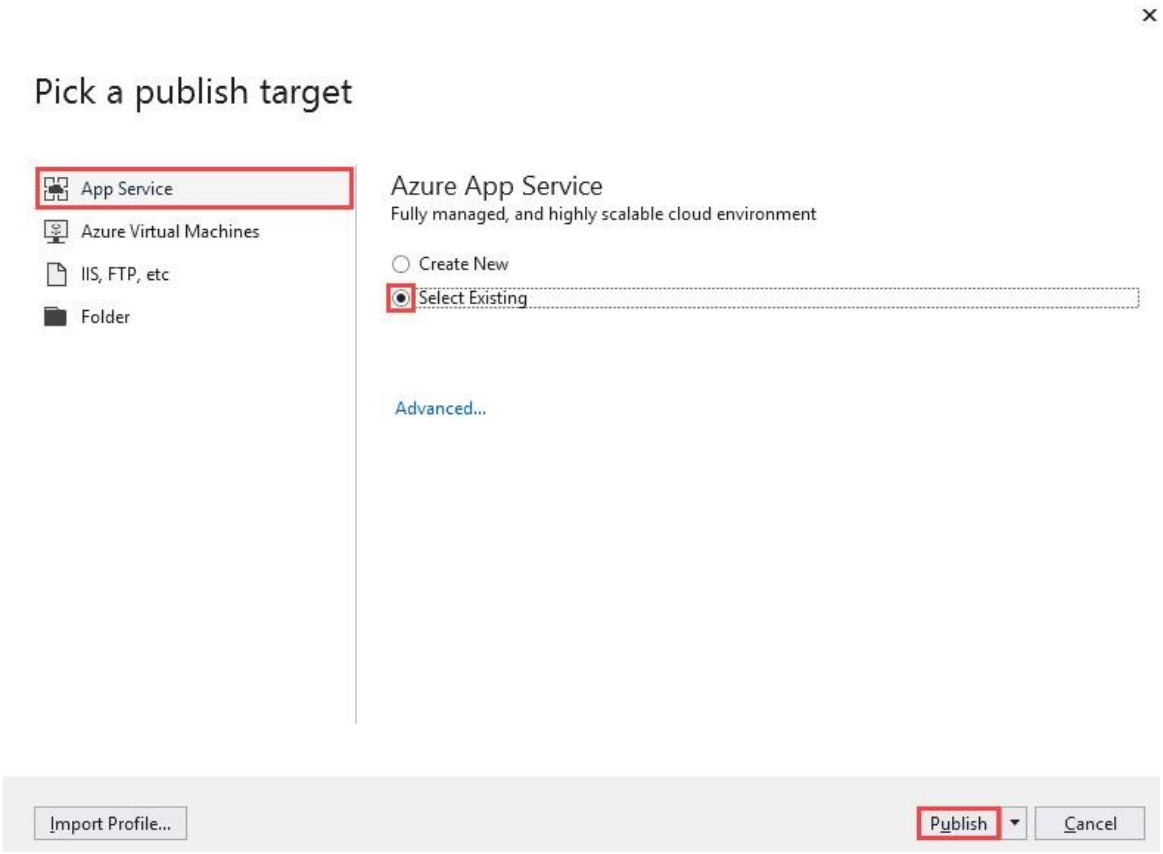
Kuva 4 Azuren käyttöönotto esimerkki

Ikkunaan (Kuva 4) syötettiin ”App name”-kenttään sovelluksen nimi tai jokin uniikki nimi, josta tuli sovelluksen verkko-osoite tosin kokonaiseen osoitteeseen Azure lisäsi oman ”. azurewebsites.net”-domaininsa. ”Subscription”-kenttään valittiin ”Azure for Students Starter”. ”Resource Group”-kentässä luotiin uusi resurssiryhmä, jolloin tulevaisuudessa voitaisiin määrittää esimerkiksi käyttöoikeuksia sovellukseen tai käyttäjille. ”App Service Plan”-kenttään lisättiin uusi palvelusuunnitelma, jolle annettiin nimi, sijainti ja hintaluokka, koska kyseessä oli sovelluskehitys, eikä verkkoliikennettä olisi paljoa sovellukseen valittiin ilmainen versio. ”SQL Database”-kentästä valittiin ”Create a new database” aukesi uusi ikkuna (Kuva 5), jossa määriteltiin tietokannan nimi, hinnoittelu ja palvelin. Koska palvelinta ei ollut vielä tehty määritettiin sille sijainti, nimi, sekä järjestelmänvalvojan tunnus ja salasana. Viimeiseksi klikattiin ”Create”-painiketta, jolloin Azure ohjasi ”Dashboard”-välilehdelle. Parin minuutin odottelun jälkeen ruutuun ilmestyi ilmoitus, että käyttöönotto onnistui ja luodut palvelut löytyivät omista osioistaan.



Kuva 5 Uuden tietokannan ja tietokantapalvelimen luominen esimerkki

Kun Azuren valmistelut oli tehty alotettiin sovelluksen julkaisemisprosessi. Visual Studiosta avattiin ”Build”-valikko, josta valittiin ”Publish”. Avautui ikkuna (Kuva 6), josta valittiin ”App Service” ja ”Select Existing”, kun klikattiin ”Publish” näppäintä avautui uusi ikkuna, josta valittiin aiemmin Azureen luotu resurssi ja painettiin ”Ok”.



Kuva 6 Sovelluksen julkaiseminen Azureen

Kun käyttöönotto oli onnistunut, avautui aiemmin määritetty verkko-osoite automaattisesti, jolloin tiedettiin sovelluksen toimivan pilvessä. Testattiin vielä, että näkymä toimi ja tietokanta oli muodostunut pilvipalvelimelle, todettiin toimivan.

6.7 Monimutkaisemman tietomallin luominen

Aiemmin oli käytetty vain yksinkertaista tietomallia, joka sisälsi vain kolme luokkaa. Tässä vaiheessa päätettiin käyttöönottaa koko tietomalli, jotta sovellusta päästäisiin oikeasti testaamaan. Vanhoihin luokkiin lisättiin ”System.CompnentModel.DataAnnotations”-nimiavaruus, jolloin pystyttiin käyttämään ominaisuuksia, joilla pystyi tekemään tarkempia määrityksiä luokkiin. ”DataType”-ominaisuutta pystyttiin käyttämään esimerkiksi sähköpostiosoitteen muuntamiseksi ”mailto:”-linkiksi, sekä päiväyksien tarkemman tietotyypin määrittämiseen. ”DataType”-ominaisuus tosin ei määrittänyt missä formaatissa päivämäärä näytettäisiin, vaan palvelimen ”CultureInfo”-luokka, joka tässä tapauksessa oli amerikkalainen. Koska sovellusta tehtiin suomalaiselle yritykselle, oli käyttäjäystävällisempää määrittää ”CultureInfo”-luokkaan suomalaiset asetukset, joka tehtiin lisäämällä luokan alustus ”Global.asax.cs”-luokan ”Application_Start()”-osioon, jolloin asetukset ajettiin sovelluksen käynnistyessä.

Samoin ”StringLenght”-ominaisuudella määrittettiin, että maatilan nimi voi olla kahdesta kolmeenkymmeneen merkkiä pitkä (Kuva 7), jos yritettiin syöttää useampia merkkejä, saatiin aikaan virheilmoitus. Samalla määritettiin sarakkeen nimi paremmin sopivaksi näkymään.

```
[Required]
[StringLength(30, MinimumLength = 2,ErrorMessage = "Nimi ei voi olla kuin 30 merkkiä pitkä")]
[Display(Name = "Maatilan nimi")]
public string Name { get; set; }
```

Kuva 7 Luokan määrityksiä

”Required”-ominaisuus määritti kentän pakolliseksi tiedoksi ja vaati lisämäärityksen ”MinimumLenght”-ominaisuuden käytön. Vastaavanlaisia määrityksiä tehtiin kaikkiin valmiina oleisiin ja tuleviin luokkiin. Määritykset olisi voinut tehdä myös samalle riville, mutta selkeämmän koodin aikaansaamiseksi jokainen ominaisuus laitettiin omalle rivilleen.

Seuraavaksi alettiin lisäämään tietomallin puuttuvat luokat, kuten aiemmin huomattiin, ettei EF pystynyt luomaan tietokantaa ”monimutkaisen” tietomallin pohjalta. Tässä mukaan astuivat ”Key” ja ”ForeignKey”-ominaisuudet, joilla EF:lle kerrottiin luokat olivat yhteydessä toisiinsa tietomallissa. ”Key”-ominaisuutta tosin ei tarvinnut käyttää, koska tietomallin jokaiseen luokkaan oli määritelty id kenttä, jollon EF sijoitti pääavaimen siihen automaattisesti. Kun tietomalli oli saatu kokonaan valmiiksi lisättiin ”Seed”-metodiin testitiedot uusiin luokkiin.

Sitten tehtiin uusi migraatio komennolla ”add-Migration ComplexDataModel”, jonka jälkeen ajettiin komento ”update-database”, jolloin EF päivitti tietokannan vastaamaan uutta tietomallia ja ajoi testitiedot tietokantaa. Toisinaan migraatioiden ajo olemassa olevaan tietokantaan, jossa oli jo tietoja. Edellä mainittu aiheutti virheilmoituksen, koska olemassa olevasta luokasta viitattiin luokkaan, josta tiedot vielä puuttuivat. Tämän ongelman korjaaminen onnistui syöttämällä ns. väliaikaisen tiedon luokkaan, johon viitattiin. Toinen mahdollinen korjausmenetelmä oli muokata ”Seed”-metodia niin ettei se yrittäisi syöttää tietoja luokkaan, joka viittaisi tyhjään luokkaan.

6.8 Viimeistelyt

Sovelluksen viimeistelynä luotiin puuttuvat näkymät kaikkiin tietomallin luokkiin, jolloin tietojen syöttäminen ja lukeminen onnistui koko tietokantaan. Testattiin uusien näkymien toiminta ja todettiin ne toimiviksi. Viimeisenä tehtiin ”yhteenveto”-näkymä, joka yhdisti kaikkien muiden näkymien tärkeimmät tiedot ja valmiiksi lasketut veroseuraamukset, kun sukupolvenvaihdos tehtäisiin suoralla kaupalla. Testattiin yhteenvedon toimivuus oikeilla maatilan tiedoilla ja saatiin yhtenevä tulos, mitä virallisessa sukupolvenvaihdos laskelmassakin oli saatu, joten todettiin toimivaksi.

7 JATKOKEHITYS

Sovellusta kehittäessä tuli vastaan paljon mielenkiintoisia ideoita, joilla sovellusta voitaisiin kehittää vielä pidemmälle. Esimerkiksi sovellukseen pitäisi lisätä yhditelmämahdollisuus perintö- ja lahjaveron osalta, jolloin sovellusta voisi käyttää kaiken kattavasti sukupolvenvaihdoksen suunnitteluun. Edellä mainittu ominaisuus mahdollistaisi sovelluksen käyttämisen suoraan tilallisella, jolloin asiantuntijan apua ei välttämättä tarvittaisi niin paljoa tai ollenkaan.

Koska sovelluksessa käsitellään henkilötietoja, pitäisi ottaa selvää uusimmista tietoturvavaatimuksista, jotta henkilötietoja käsiteltäisiin säädöksien vaatimalla tavalla. Tietoturvaan pitäisi muutenkin panostaa lisää, koska sovellus toimii julkisella pilvipalvelimella ei oikeita tietoja ole turvallista käyttää laskemiseen.

Kaupallisessa mielessä pitäisi miettiä onko alkuperäisellä sovelluksen tilaajalla vielä kiinnostusta kyseiseen sovellukseen vai onko projekti jo unohtunut. Sovelluksen voisi myydä kokonaisuudessaan yritykselle tai pohtia sovelluksen lisenssointia, jolloin sen saisi käyttöön useammalle yritykselle tai suoraan tilallisille.

Sovelluksen voisi myös pienillä muutoksilla valjastaa muidenkin kuin maatilojen sukupolvenvaihdoksien veroseuraamusten suunnitteluun. Veroseuraamukset kuitenkin lasketaan samanlailla, mutta maatilojen sukupolvenvaihdoksiin vaadittavan informaation määrä on huomattavasti suurempi kuin normaalikokoisen perheyrityksen. Sovellusta voisi käyttää myös perunkirjoituksiin, jotka yleensä tehdään pankin asiantuntijoiden avulla.

8 POHDINTA

Opinnäytetyöksi Spv-sovelluksen kehitys tuntui hyvältä ratkaisulta, koska projekti tarjosi hyvät mahdollisuudet kehittää osaamistani aihealueen sisällä. Koulutusohjelmani tarjosi hyvät lähtötiedot ja valmiudet toteuttaa kyseinen projekti, kuitenkin projektin onnistunut suorittaminen vaati paljon itseopiskelua.

Projektin aikana opin paljon uusia menetelmiä, sekä miten suunnitella web-client sovelluskehitys huomattavasti paremmin, virheistä oppii.

Projektissa mielestäni onnistui sovelluksen toteutus, koska lopputuloksena oli toimiva sovellus, jolla pystyttiin tekemään sovellukselta vaadittu funktio. Koko opinnäyteprojektissa pahiten pieleen meni aikataulutus, sekä sovelluksen suunnitteluun käytetty aika.

Vastaavan projektin aloituksessa ei pitäisi kuluttaa liikaa aikaa sovelluksen suunnitteluun vaan aloitta toteutus heti ja ratkoa ongelmia sitä mukaa kun niitä ilmenee, eikä puida turhaan mahdollisia ongelmia, joita ei välttämättä koskaan tapahdukkaan.

Projektissa vastaan tuli paljon haasteita, joiden ratkaiseminen toi myös paljon onnistumisen tunnetta, joiden voimalla sovellus päätyi viimeisimpään muotoonsa.

9 LÄHDELUETTELO

Helsingin yliopisto. *SQL history*. Haettu 1. 5. 2020 osoitteesta https://www.cs.helsinki.fi/u/laine/tuelip/sql_material/sql_history.html

Microsoft. *ASP.NET MVC Pattern*. Haettu 9. 12. 2019 osoitteesta https://dotnet.microsoft.com/apps/aspnet/mvc

Microsoft. *Microsoft Azure*. Haettu 15. 3. 2020 osoitteesta https://azure.microsoft.com/en-us/overview/

Sivonen, V.-M. (4. 4. 2004). *C#-kieli*. Haettu 15. 3. 2020 osoitteesta https://www.cs.helsinki.fi/u/pohjalai/k04/ohpe/seminar/Sivonen-CSharp.pdf

W3C. *HTML*. Haettu 15. 3. 2020 osoitteesta https://www.w3.org/standards/